TEZPUR UNIVERSITY

Bachelor of Technology

Computer Science and Engineering

Project Report on

Build Your Own User-Level File System

Submitted By

SOUMYASISH SARKAR, CSB22054

Under the guidance of

Prof. Sanjib Kumar Deka

Professor, CSE, TU

Spring 2025

Department of Computer Science and Engineering

TEZPUR UNIVERSITY

Tezpur, 784028

Tezpur University

School of Engineering, Tezpur, 784028, Assam, India

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the project entitled **"Build Your Own User-Level File System"** is a bonafide work carried out by **Soumyasish Sarkar (Enrollment No.: CSB22054)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering**, under the course code **CO317 – Project - I(Using SE Perspective)**, during the **Spring Semester 2025**.

The project has been carried out using a Software Engineering perspective, under the guidance of **Prof. Sanjib Kumar Deka**, Professor, Department of Computer Science and Engineering, Tezpur University.

Prof. Sanjib Kumar Deka	**Dr. S. Ibotombi Singh**	**Dr. Sarat Saharia**
Professor	Project Co-ordinator	Professor & HOD
Dept. of CSE	Dept. of CSE	Dept. of CSE
School of Engineering	School of Engineering	School of Engineering
Tezpur University	Tezpur University	Tezpur University
Signature	Signature	Signature .....

Name of Examiners:	**Signature with Date:**
1. _____	_____
2. _____	_____
3. _____	_____

Acknowledgement

The completion of this project would not have been possible without the support of several individuals. I would like to express my sincere gratitude to all those who assisted me throughout this work.

Firstly, I extend my heartfelt thanks to Prof. Sanjib Kumar Deka, my project guide, for his invaluable guidance, support, and expertise. His encouragement shaped the project in the right direction.

I am also deeply grateful to Dr. Sarat Saharia, Professor and Head of the Department of Computer Science and Engineering, Tezpur University, for his continuous support and motivation, which pushed me to extend my academic boundaries.

My thanks go to Dr. S. Ibotombi Singh, the Project Coordinator, and all the faculty members of the Department of Computer Science and Engineering, Tezpur University, for their constructive feedback and encouragement.

Special thanks to the Tezpur University School of Engineering for providing the essential infrastructure and resources that were crucial to completing the project successfully.

I also acknowledge the contributions of my classmates and friends, whose support and valuable insights helped me through various stages of the project.

Lastly, I am profoundly grateful to my family for their unwavering support and encouragement, which kept me moving forward through the challenges.

I also appreciate anyone else who directly or indirectly contributed to the success of this project.

Soumyasish Sarkar

CSB22054

Abstract

project explores the design and development of a custom kernel-level file system that can be controlled via a user-friendly web-based interface. Unlike traditional FUSE-based user-level systems, our solution operates within the Linux kernel, offering low-level control over file creation, reading, writing, directory management, linking, and journaling. The frontend UI, developed using HTML, JavaScript, and Socket.IO, enables real-time interaction with the mounted file system, enhancing user accessibility and visibility into kernel-level operations. The backend uses Node.js to safely execute privileged system commands that interact with the custom file system. This project serves as a bridge between system-level programming and user-centric interaction, demonstrating key OS concepts like inodes, file descriptors, access permissions, journaling, and modular kernel design.

Contents

1 Introduction

1.1 Scope

This project implements a minimal kernel-level file system and integrates it with a user-space web-based UI. It supports file and directory creation, write to a file, read a file, delete file and directory, hard link and symbolic links, custom permission checks, journaling, and basic user interaction via an HTML interface.

1.2 Purpose of the Project

The main purpose is to bridge the gap between kernel-level file system operations and user interaction. Most file systems are either handled by the OS or created using user-space FUSE. This project demonstrates the construction of a real kernel module file system with frontend control. This file system will interact directly with the kernel, providing a more efficient, low-level interface for file operations. It aims to expand the student's understanding of operating system concepts, particularly file system architecture and kernel development. This also aims to develop a foundation to integrate journaling mechanisms and ML-based optimization in the future.

1.3 Overview of the Report

This report details the background, feasibility, objectives, architecture, flow diagrams, code structure, features, limitations, and results of the project. The approach, key modules, user interface logic, and system interactions are all covered.

2 Problem Statement

Existing file systems in Linux are either complex or abstracted via FUSE. This prevents students from learning real file system internals like inode management, permissions, journaling, and device mounting. We need an accessible yet low-level file system development model with user-level interaction.

This project proposes the design and implementation of a kernel-level file system to bridge this gap. By working at the system level, the project will:

Enable direct control over file handling and metadata structures.

Provide hands-on experience with kernel-level system calls and data structures.

Offer insights into secure memory management and modular design.

Establish a foundation for advanced features such as journaling, access control, and AI-based optimization.

This project proposes a kernel-level file system design to:

— Enable low-level control over file handling and metadata.

— Provide firsthand experience with kernel system calls.

— Create a platform for future enhancements like AI-based optimization.

3 Software Requirements Specification (SRS)

3.1 Functional Requirements

- **FR1:** The system shall allow creation of files and directories.
- **FR2:** The system shall support reading from and writing to files.
- **FR3:** The system shall enable creation of hard and symbolic links.
- **FR4:** The system shall check file permissions based on user and group.
- **FR5:** The system shall maintain a journal of file creation and deletion.
- **FR6:** The system shall display output and status messages on a web UI.
- **FR7:** The system shall provide access to kernel logs in real-time.

3.2 Non-Functional Requirements

- **NFR1:** The system shall respond to file operations within 1 second.
- **NFR2:** The UI shall be responsive and mobile-friendly.
- **NFR3:** The system shall support concurrent user requests.
- **NFR4:** The system must be secure and prevent unauthorized command execution.

3.3 System Requirements

- Linux OS with kernel module support.
- Node.js environment for backend.
- Browser to access frontend UI.

4 Data Flow Diagrams (DFD)

4.1 Level 0 DFD

Figure 1: Level 0 Data Flow Diagram

4.2 Level 1 DFD

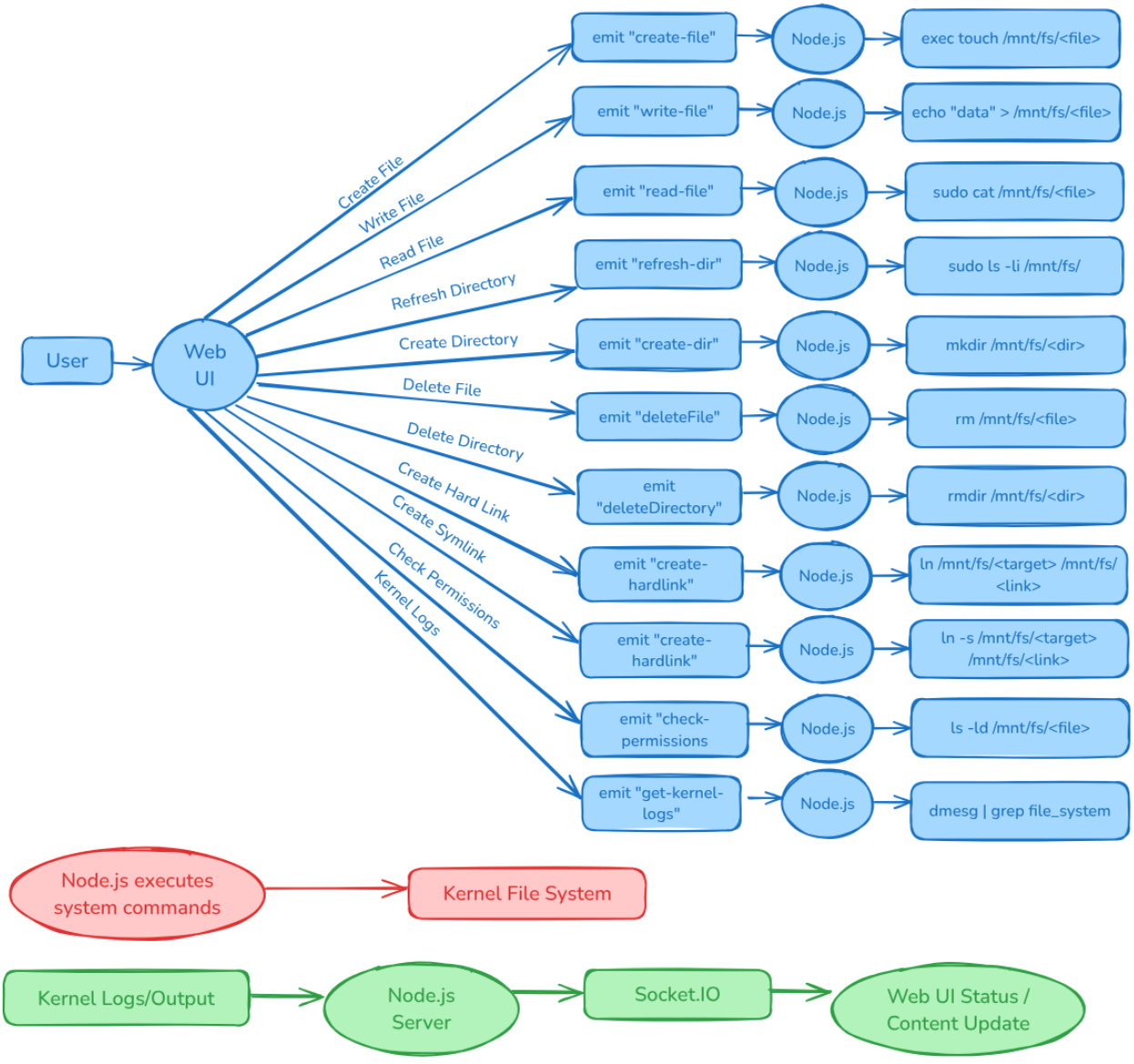

Figure 2: Level 1 Data Flow Diagram

5 Objectives

1. **Implement a custom file system as a kernel module:**
    - A Linux kernel module was developed to register and mount a new file system named `file_system`. It defines its own superblock, inode, and file operations.
2. **Support file operations: open(), read(), write(), unlink():**
    - The file system implements `file_open`, `file_read`, `file_write`, and `file_unlink` with permission checks and journaling support for create/delete operations.
3. **Directory operations: mkdir(), rmdir(), create(), lookup():**
    - Custom directory handling is implemented with `file_mkdir`, `simple_rmdir`, `file_create`, and `file_lookup`, supporting hierarchical structure and metadata management.
4. **Implement symbolic and hard links:**
    - Symbolic links are created using a custom `file_symlink()` with dynamic memory for link targets, while hard links are supported through `simple_link`.
5. **Enforce access control using permissions:**
    - A custom `check_permissions()` function validates access rights based on user, group, and others against inode permission bits before allowing operations.
6. **Inode-Based File System Structure:**
    - Inodes are dynamically created using `file_system_make_inode()` and include file type, ownership, link count, and function pointers for operation dispatching.
7. **Add journaling for create/delete events:**
    - A circular journaling system logs file creation and deletion using `journal_start()` and `log_journal_entry()` to enhance traceability and debugging.
8. **Build a responsive HTML UI to trigger kernel-level operations:**
    - A dynamic frontend built with HTML, CSS, and JavaScript connects to a Node.js backend, allowing users to invoke kernel-level file operations interactively.
9. **Provide logs and outputs to users:**

- Kernel logs with the prefix `"file_system:"` are fetched via `dmesg` and displayed on the UI every 5 seconds to inform users of system activity.

10. **Simulate a file content system with file data[]:**

    - A statically allocated buffer `file_data[]` simulates file contents, allowing read/write access with size bounds and default content initialization.

11. **Modular and extensible architecture:**

    - The system is modular, with clean separation of file ops, directory ops, journaling, and UI code, enabling future extensibility and maintainability.

6 Feasibility Study

1. **Technical Feasibility**

   - Kernel Module Support: Modern Linux distributions support dynamically loadable kernel modules.
   - Frontend (HTML/CSS/JS): Standard browser technologies for user interaction.
   - Backend (Node.js + Socket.IO): Executes shell commands for system-level operations.

2. **Economic Feasibility**

   - Open-source tools used: no software licensing costs.
   - Requires standard Linux setup only.

3. **Operational Feasibility**

   - Demonstrates real-world OS concepts.
   - Easy to extend or simulate advanced operations like journaling and access control.

7 Background Study

- Linux VFS layer
- Superblock, inode, dentry, and `file_operations` structures
- Permissions model (UID/GID/`i_mode`)
- Kernel space vs user space
- Node.js child processes and Socket.IO

8 Proposed Idea

1. A custom file system is implemented in kernel space and registered with a unique magic number and type. User requests are routed via a web UI (built with HTML + JS), which sends actions to a Node.js backend. The backend triggers system-level commands like touch, mkdir, ln, cat, rm, which operate on a mounted directory managed by the custom kernel module.

9 System Architecture

9.1 Subsystems and Components

9.1.1 Kernel Module

- **File operations:** `open`, `read`, `write`
- **Directory operations:** `create`, `lookup`, `mkdir`, `unlink`, `symlink`
- **Journaling:** `journal_start`, `log_journal_entry`
- **Permission logic:** `check_permissions()`

9.1.2 User Interface

- **HTML/CSS Frontend:** Forms and buttons for user interaction
- **JavaScript (Socket.IO):** Sends user actions to the backend

9.1.3 Backend

- **Node.js Server:** Executes shell commands with `sudo`
- **Status Handling:** Emits status and log messages to the client

9.2 Interaction Flow

- **User clicks button on UI:** User interacts with the interface by clicking a button.
- **JavaScript triggers `socket.emit()`:** JavaScript sends parameters to the backend using `socket.emit()`.
- **Backend runs `sudo` system command:** The backend executes a system command with `sudo`.
- **Kernel module handles the operation:** The kernel module processes the operation using kernel-level functions.
- **Logs printed using `printk()`:** Logs are captured and displayed through `printk()` for debugging.
- **Results shown on browser:** The operation results are sent back to the frontend and displayed in the browser.

10 Detailed Flowcharts

10.1 6.1 File Operation Flow (Read/Write)

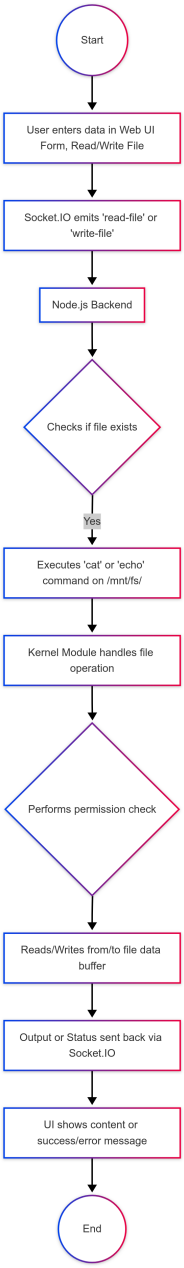

Figure 3: File Operation Flow (Read/Write)

10.2 6.2 Mounting and Unmounting Flow

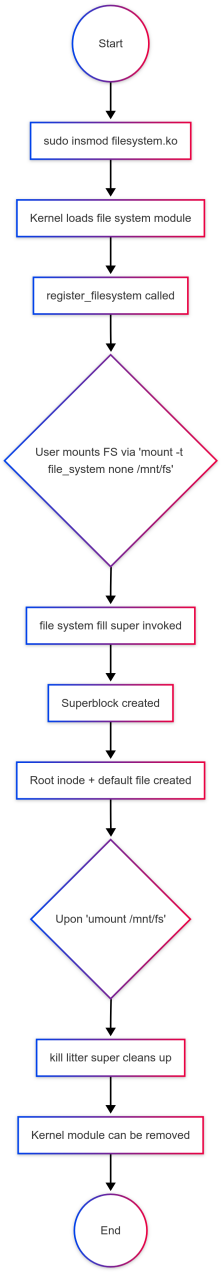

Figure 4: Mounting and Unmounting Flow

10.3 6.3 Linking Mechanism (Hard/Symbolic)

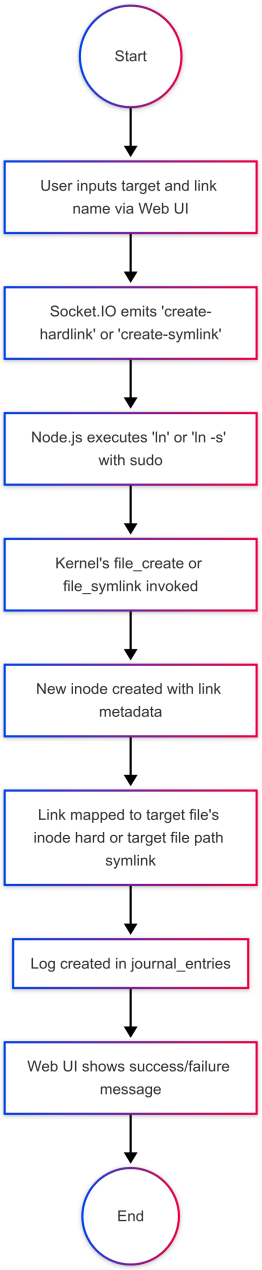

Figure 5: Linking Mechanism (Hard/Symbolic)

10.4 6.4 Delete File/Directory Flow

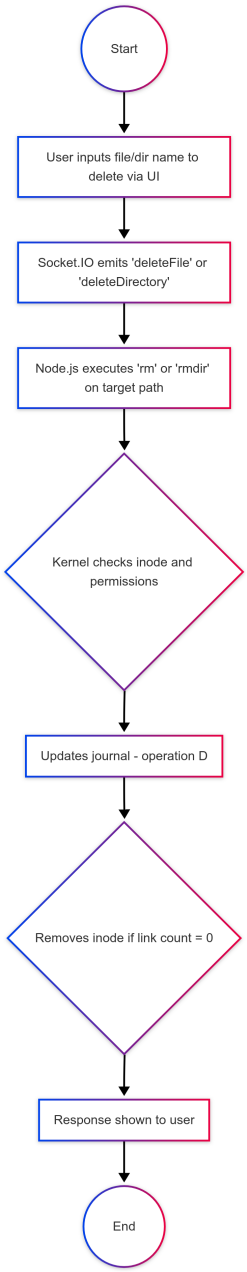

Figure 6: Delete File/Directory Flow

10.5 6.5 Directory Listing and Refresh

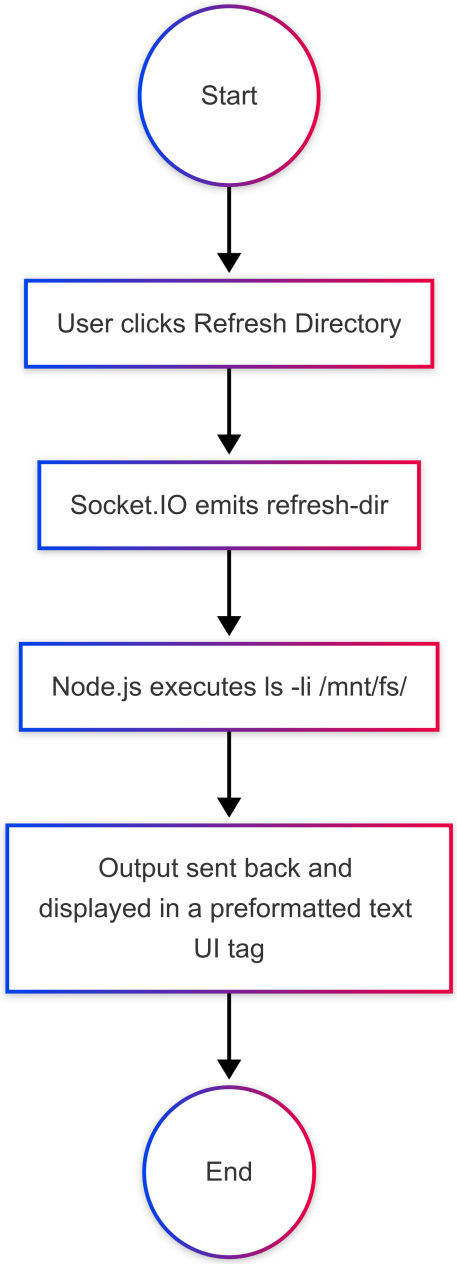

Figure 7: Directory Listing and Refresh

10.6 6.6 Check Permissions Flow

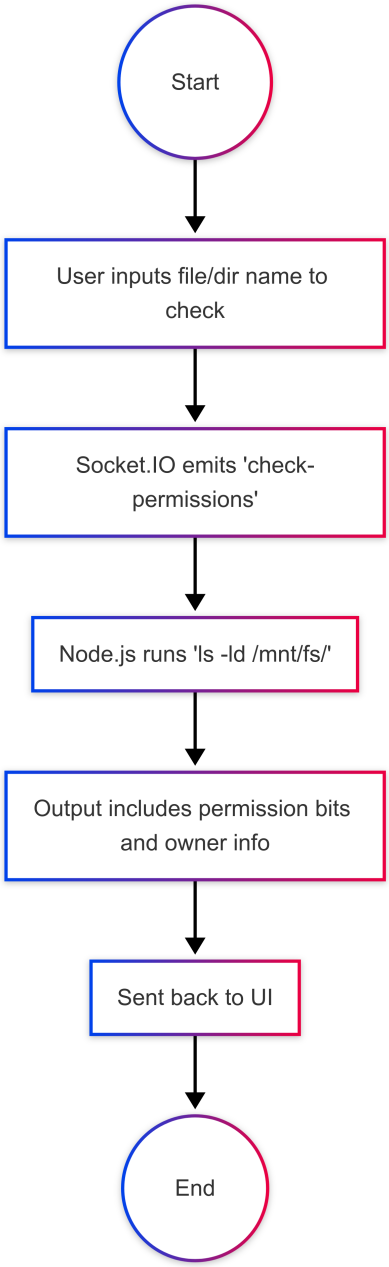

Figure 8: Check Permissions Flow

10.7 6.7 Status and Log Display

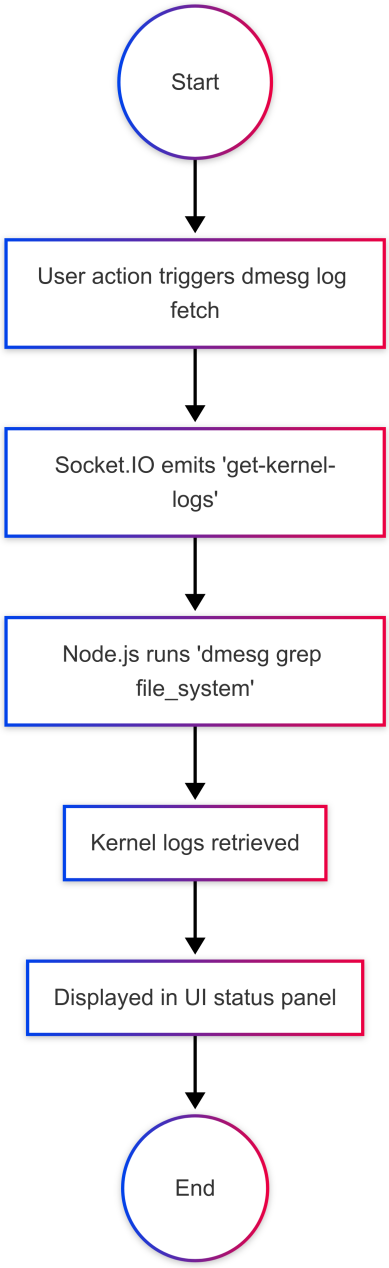

Figure 9: Status and Log Display

11 Code Structure

11.1 Superblock

- **Magic Number:** 0x1CEB00DA
- **Superblock registered on mount.**

11.2 File Operations

- **file_open:** Permission check + default content setup.
- **file_read:** Reads from `static file_data[]`.
- **file_write:** Appends to `file_data[]`.

11.3 Inode Operations

- **file_create, file_symlink, file_unlink:** Operations on inodes.
- **Permission check before every operation.**
- **Links maintained using `inode_inc_link_count`.**

11.4 Journaling

- Logs operations 'C' (create) and 'D' (delete).
- Uses `journal_entries[MAX_JOURNAL_ENTRIES]` for maintaining the journal.

12 Permissions and Access Control

- **Permissions based on current UID/GID.**
- **i_mode** used to check for read, write, and execute permissions.
- `check_permissions()` used inside all operations to verify access rights.
- Follows the UNIX `rwx` pattern for permission settings.

13 High-Level Features Implemented

Feature	**Description**
File system registration	Registers with VFS using `file_system_name`
Superblock & inode	Created on mount using `file_system_make_inode()`
File ops	`open()`, `read()`, `write()` supported
Dir ops	`create`, `mkdir`, `unlink`, `symlink`
Hard links	`simple_link()` used
Symbolic links	Custom `file_symlink()` function
Journaling	Logs using `journal_entry` struct
Permissions	Owner/group/others permission checked
Default content	`file_open()` adds default text

14 Limitations

- **File contents are stored in one global buffer (not scalable).**
- **No persistent journaling (in-memory only).**
- **Limited permission model (basic ACL).**
- **No caching or file metadata tracking.**

15 Future Work

- **Persistent journaling to disk.**
- **Integrate inode-to-content mapping for multiple files.**
- **Add GUI-based permission editing.**
- **File system benchmarking and stress testing.**
- **AI/ML-based access pattern tracking and optimization.**

16 Conclusion

This project successfully demonstrates the creation of a Linux kernel file system with a web-based UI for user interaction. It integrates key OS concepts like inodes, journaling, linking, and permission enforcement. The modular design allows future expansion with persistent logging, advanced ACLs, and even AI-assisted enhancements.

17 References

References

[1] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
`https://archive.org/details/designofunixoper00bach`

[2] Robert Love, *Linux Kernel Development*, 3rd Edition, Addison-Wesley, 2010.

[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, 3rd Edition, O'Reilly Media, 2005.
`https://lwn.net/Kernel/LDD3/`

[4] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, 3rd Edition, O'Reilly Media, 2005.
`https://www.oreilly.com/library/view/understanding-the-linux/0596005652/`

[5] Micah Elizabeth Scott, "A Simple Filesystem from Scratch in Linux Kernel Space (LWN Article)", 2018.
`https://lwn.net/Articles/659523/`

[6] Ruwaid Louis and David Yu, "A study of the exploration/exploitation trade-off in reinforcement learning: Applied to autonomous driving", 2019.
`https://www.diva-portal.org/smash/get/diva2:1336430/FULLTEXT01.pdf`

[7] *Linux Kernel Docs: filesystems.txt, inode.c*

[8] *LWN.net articles on VFS*

[9] *Socket.IO documentation*

[10] *The Linux Programming Interface by Michael Kerrisk*

18 Appendix

18.1 Github Code Link

The complete implementation of the project is available on GitHub. You can access the code repository using the following link:

https://github.com/soumyasish-sarkar/file-system

18.2 Sample Commands

```
$ make clean
$ make
$ sudo insmod filesystem.ko
$ sudo mkdir -p /mount/fs
$ sudo mount -t file_system none /mont/fs
$ sudo ls -li /mount/fs
$ sudo dmesg | grep "file_system"
$ sudo ln /mount/fs/myfile /mount/fs/hfile
$ sudo ln -s /mouny/fs/myfile /mount/fs/sfile
$ sudo ls -li /mount/fs
$ touch /mnt/fs/myfile
$ cat /mnt/fs/myfile
```

```
soumyasish@sarkar:~/programing/file-system$ make clean
make -C /lib/modules/6.11.0-25-generic/build M=/home/soumyasish/programing/file-system clean
make[1]: Entering directory '/usr/src/linux-headers-6.11.0-25-generic'
make[1]: Leaving directory '/usr/src/linux-headers-6.11.0-25-generic'
soumyasish@sarkar:~/programing/file-system$ make
make -C /lib/modules/6.11.0-25-generic/build M=/home/soumyasish/programing/file-system modules
make[1]: Entering directory '/usr/src/linux-headers-6.11.0-25-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
  You are using:           gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
  CC [M]  /home/soumyasish/programing/file-system/filesystem.o
  MODPOST /home/soumyasish/programing/file-system/Module.symvers
  CC [M]  /home/soumyasish/programing/file-system/filesystem.mod.o
  LD [M]  /home/soumyasish/programing/file-system/filesystem.ko
  BTF [M] /home/soumyasish/programing/file-system/filesystem.ko
Skipping BTF generation for /home/soumyasish/programing/file-system/filesystem.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.11.0-25-generic'
soumyasish@sarkar:~/programing/file-system$ sudo insmod filesystem.ko
soumyasish@sarkar:~/programing/file-system$ sudo mkdir -p /mount/fs
soumyasish@sarkar:~/programing/file-system$ sudo mount -t file_system none /mount/fs/
soumyasish@sarkar:~/programing/file-system$ S
```

Figure 10: Kernel commands

18.3 Kernel Debug Logs

```
soumyasish@sarkar:~/programing/file-system$ sudo dmesg | grep "file_system"
[  654.623748] file_system: Registering file system
[  665.491990] file_system: mounting...
[  665.492008] file_system: inode created with mode 40755
[  665.492014] file_system: inode created with mode 100644
[  665.492016] file_system: superblock initialized with 'myfile'
[  764.805324] file_system: unmounting...
[  787.112010] file_system: Unregistering file system
[  803.632226] file_system: Registering file system
[  826.667772] file_system: mounting...
[  826.668073] file_system: inode created with mode 40755
[  826.668078] file_system: inode created with mode 100644
[  826.668079] file_system: superblock initialized with 'myfile'
[ 1079.697780] file_system: inode created with mode 100644
[ 1079.697786] file_system: Journal started for inode 32940, operation C
[ 1079.697790] file_system: Journaled operation 'C' for inode 32940 (Txn 2)
[ 1079.697791] file_system: Created file 'file'
[ 1079.697806] file_system: Default content added to file
[ 1079.697806] file_system: file_open allowed
[ 2058.696058] file_system: unmounting...
[ 2066.694857] file_system: Unregistering file system
[ 7876.734476] file_system: Registering file system
[ 7878.864596] file_system: mounting...
[ 7878.864615] file_system: inode created with mode 40755
[ 7878.864620] file_system: inode created with mode 100644
[ 7878.864622] file_system: superblock initialized with 'myfile'
soumyasish@sarkar:~/programing/file-system$
```

Figure 11: Kernel Debug Logs

18.4 Terminal Output Screenshots

18.4.1 File Present

```
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
59871 -rw-r--r-- 2 root root 0 May 15 02:30 myfile
```

Figure 12: Listing the file present in the mounted directory.

18.4.2 Hard Link Creation

```
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
59871 -rw-r--r-- 2 root root 0 May 15 02:30 myfile
soumyasish@sarkar:~/programing/file-system$ sudo ln /mount/fs/myfile /mount/fs/hardfile
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
59871 -rw-r--r-- 3 root root 0 May 15 02:30 hardfile
59871 -rw-r--r-- 3 root root 0 May 15 02:30 myfile
```

Figure 13: Creating a hard link to the existing file.

18.4.3 Symbolic Link Creation

```
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
59871 -rw-r--r-- 3 root root 0 May 15 02:30 hardfile
59871 -rw-r--r-- 3 root root 0 May 15 02:30 myfile
soumyasish@sarkar:~/programing/file-system$ sudo ln -s /mount/fs/myfile /mount/fs/softfile
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
59871 -rw-r--r-- 3 root root  0 May 15 02:30 hardfile
59871 -rw-r--r-- 3 root root  0 May 15 02:30 myfile
    ? lrwxrwxrwx 2 root root 16 May 15 02:49 softfile -> /mount/fs/myfile
```

Figure 14: Creating a symbolic (soft) link.

18.4.4 Verifying File, Hard Link, and Soft Link

```
soumyasish@sarkar:~/programing/file-system$ cat /mount/fs/myfile
Content of myfile
soumyasish@sarkar:~/programing/file-system$ cat /mount/fs/hardfile
Content of myfile
soumyasish@sarkar:~/programing/file-system$ cat /mount/fs/softfile
Content of myfile
```

Figure 15: Displaying content of original file, hard link, and symbolic link using `cat`.

18.4.5 File Creation Using touch

```
soumyasish@sarkar:~/programing/file-system$ sudo touch /mount/fs/file
soumyasish@sarkar:~/programing/file-system$ sudo ls -li /mount/fs
total 0
60983 -rw-r--r-- 2 root root  0 May 15 02:54 file
59871 -rw-r--r-- 3 root root  0 May 15 02:30 hardfile
59871 -rw-r--r-- 3 root root  0 May 15 02:30 myfile
    ? lrwxrwxrwx 2 root root 16 May 15 02:49 softfile -> /mount/fs/myfile
```

Figure 16: Creating a new file in the mounted directory using `touch`.

18.5 UI Interface Output Screenshots

18.5.1 Node.js Starting

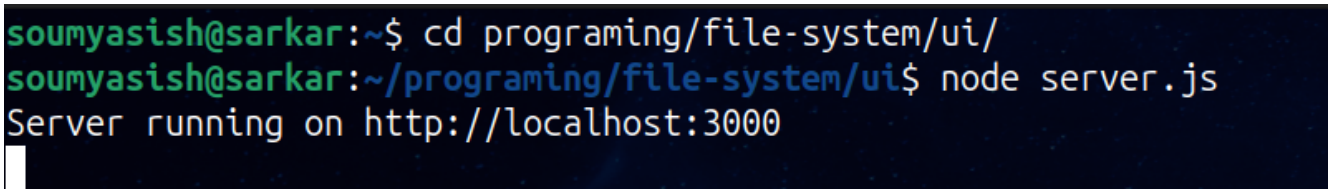

Figure 17: Node.js starting.

18.5.2 Home Page

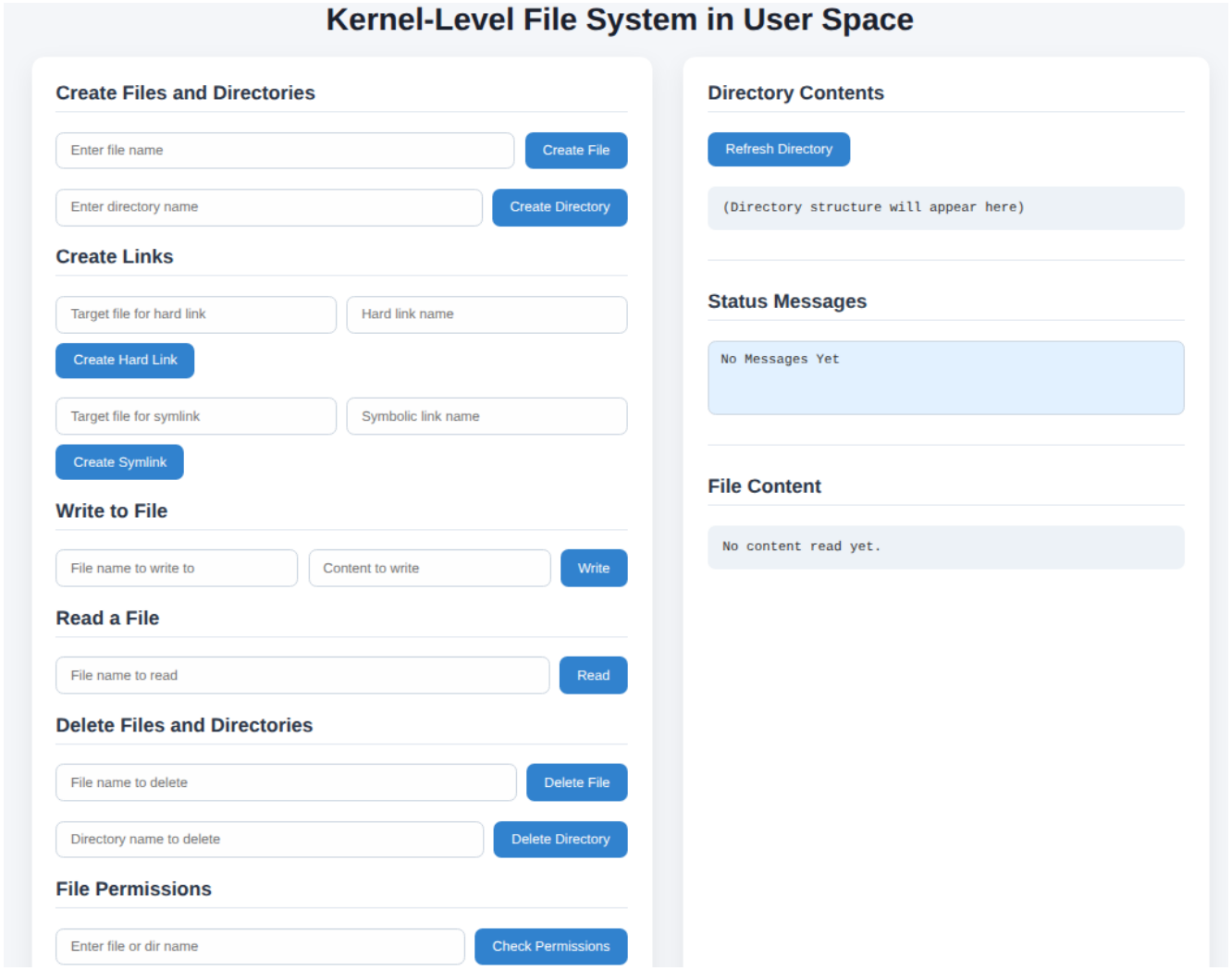

Figure 18: Home page of the application.

18.5.3 Directory Content Refresh

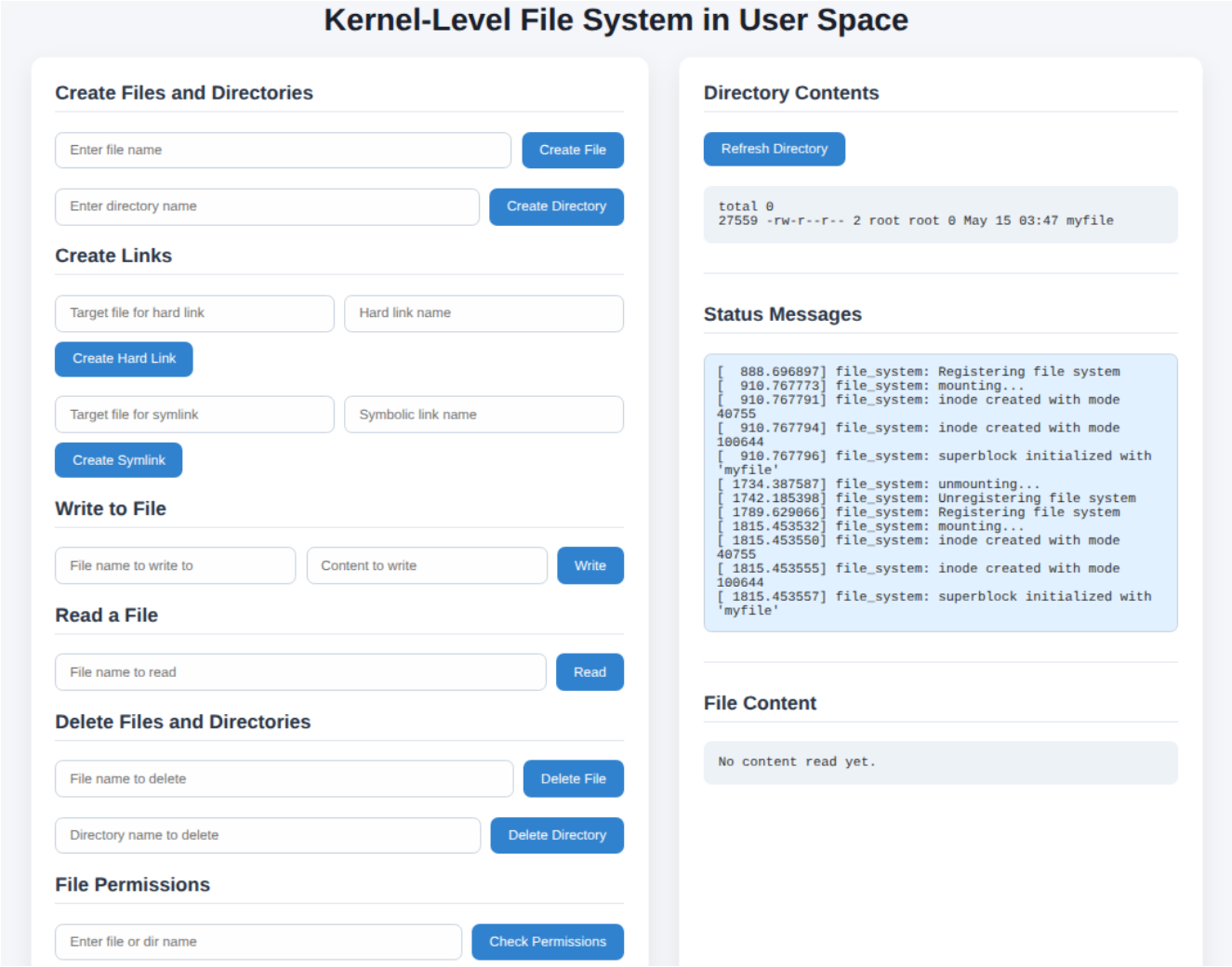

Figure 19: Directory content after refresh.

18.5.4 Create File Status Message

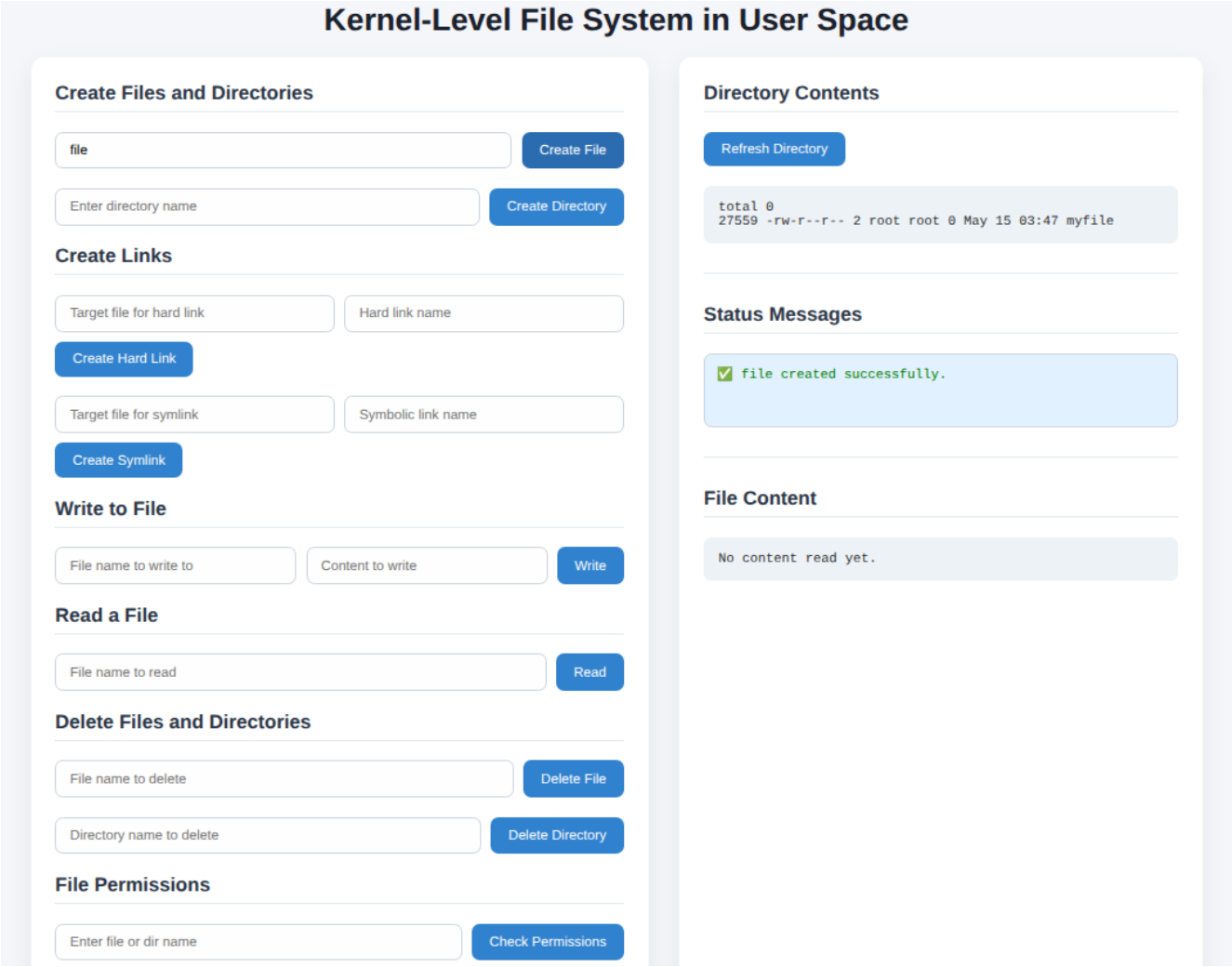

Figure 20: Status message after file creation.

18.5.5 Directory Refresh After File Creation

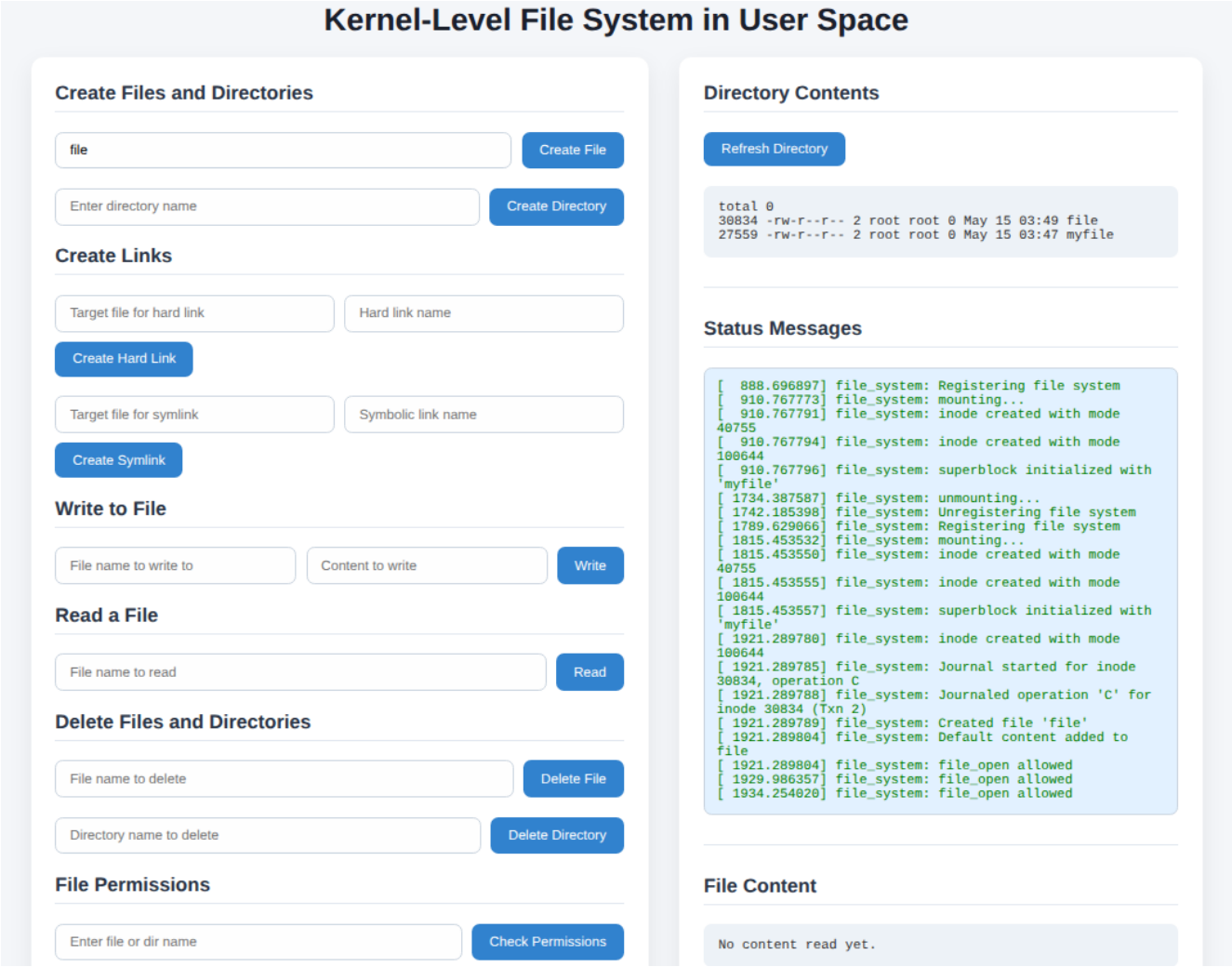

Figure 21: Directory content after file creation and refresh.

18.5.6 Directory Creation Status Message

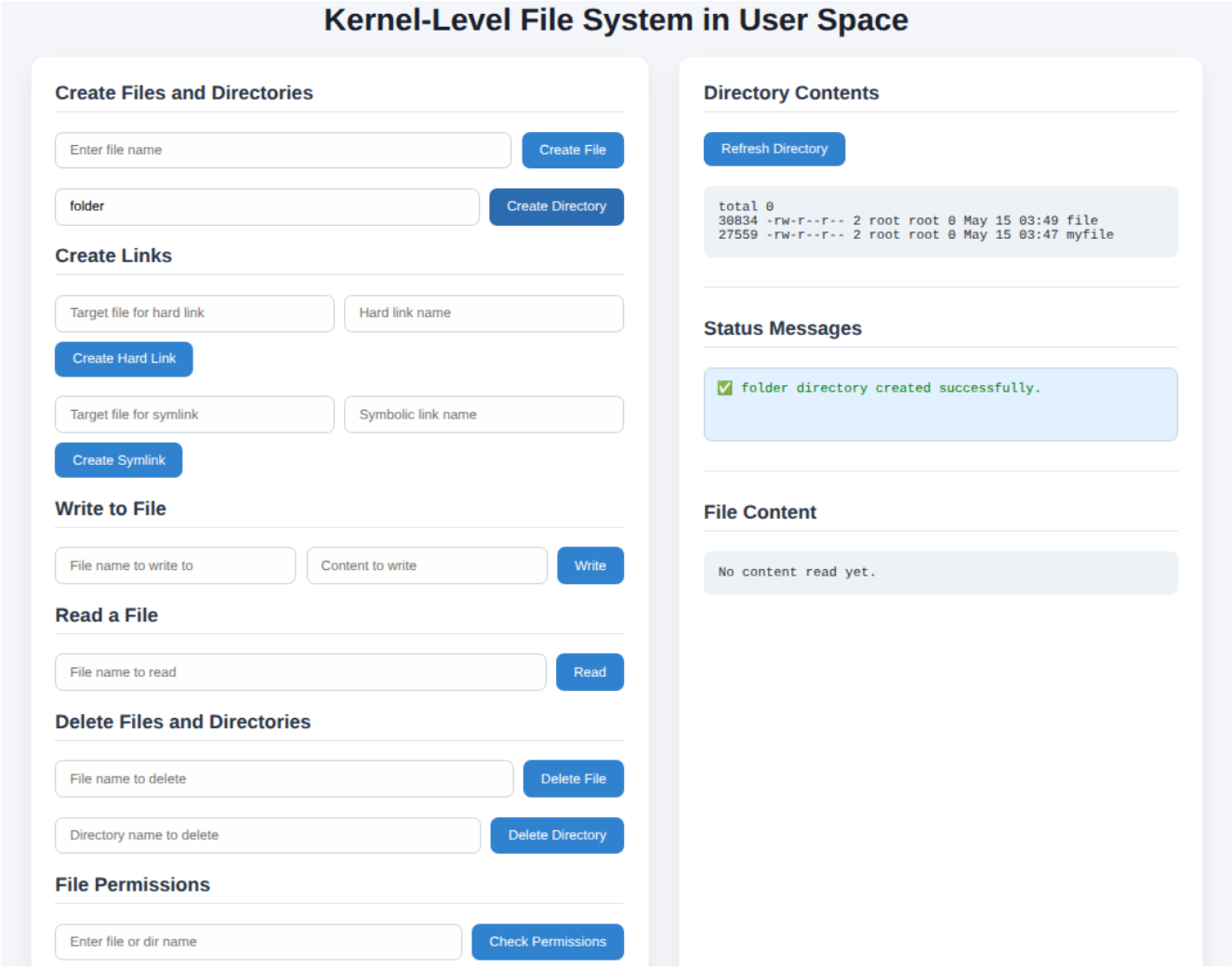

Figure 22: Status message after directory creation.

18.5.7 Directory Content Refresh After Directory Creation

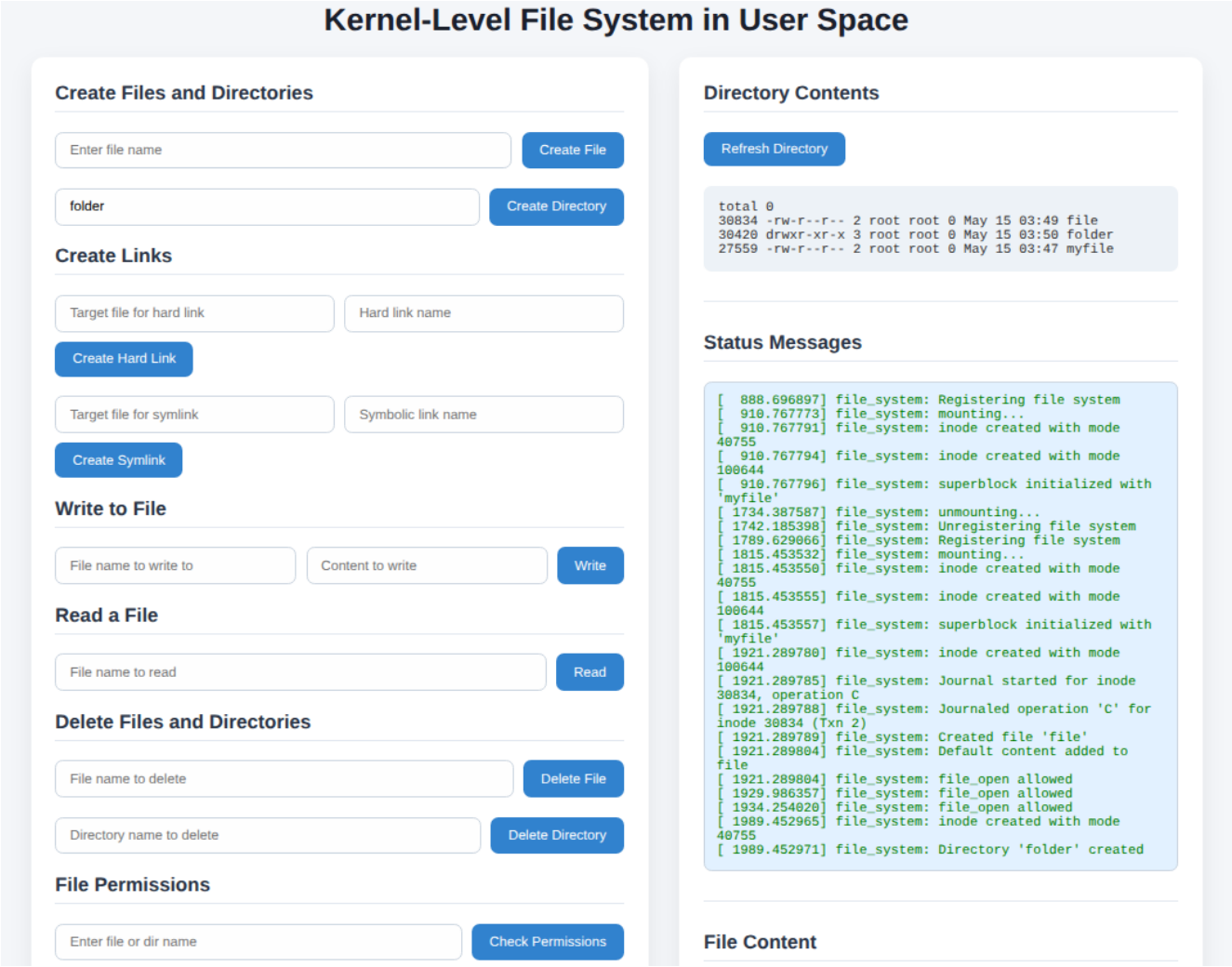

Figure 23: Directory content after directory creation and refresh.

18.5.8 Hard Link Create Status Message

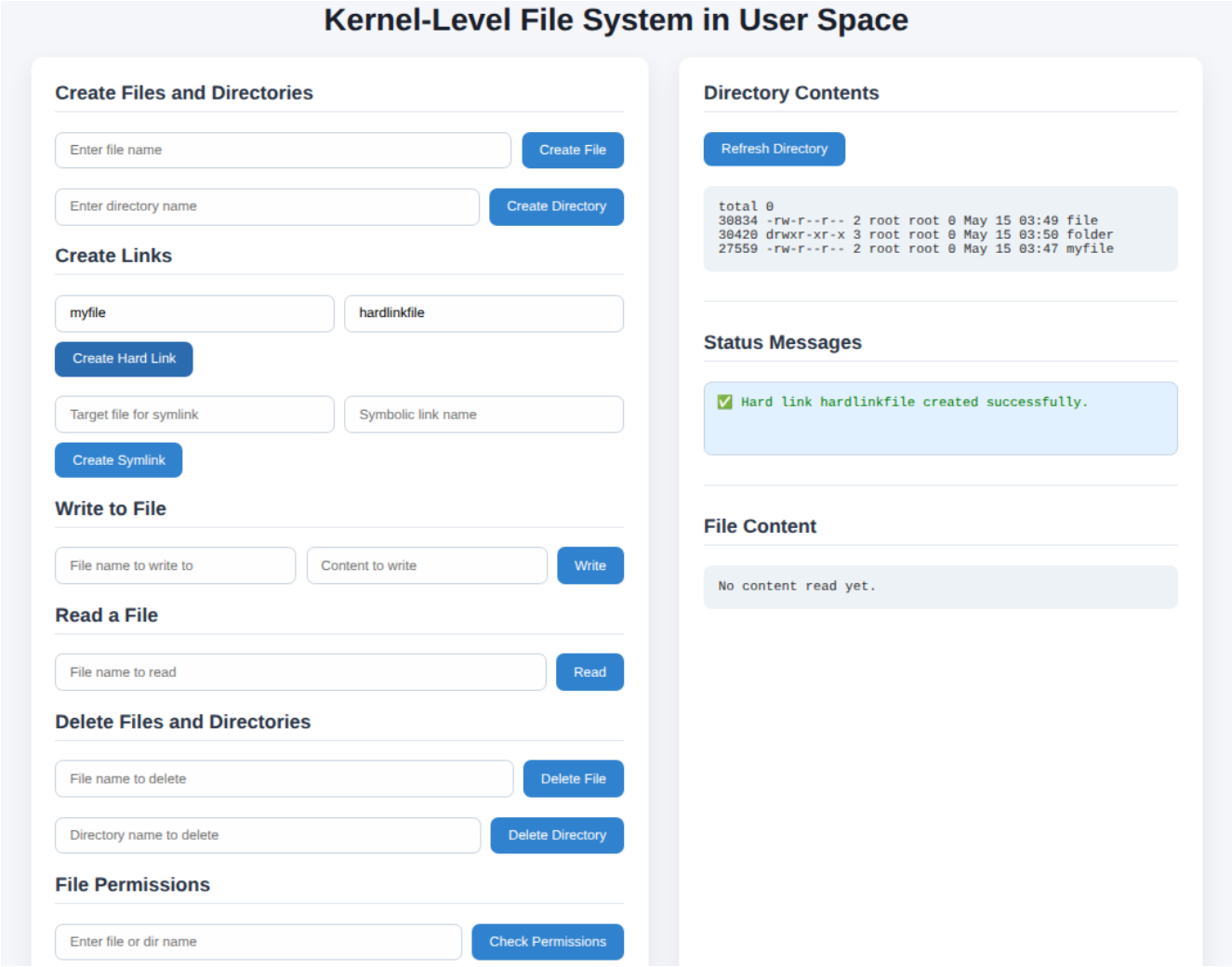

Figure 24: Status message after hard link creation.

18.5.9 Soft Link Create Status Message

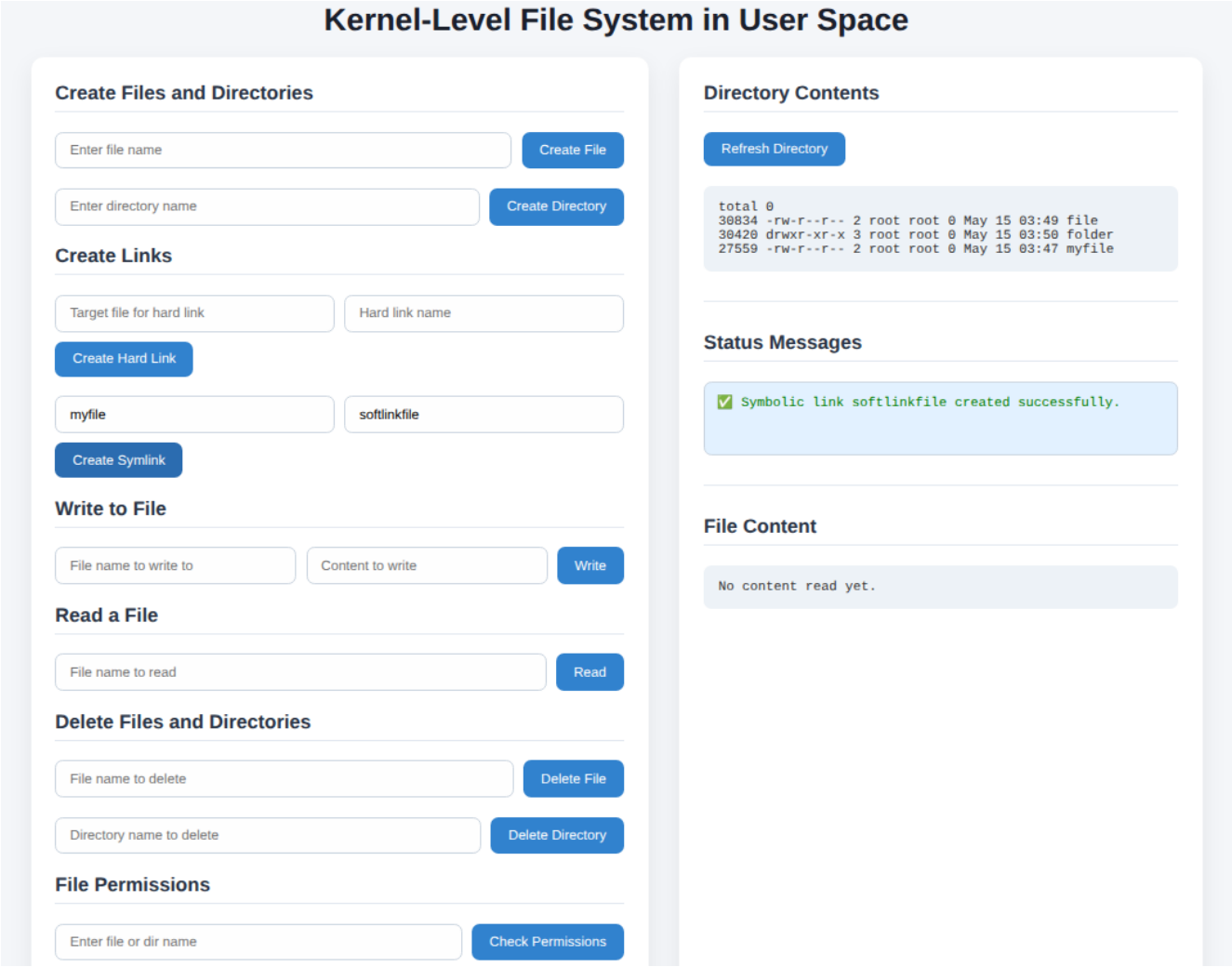

Figure 25: Status message after soft link creation.

18.5.10 Directory Content Refresh After Soft Link Creation

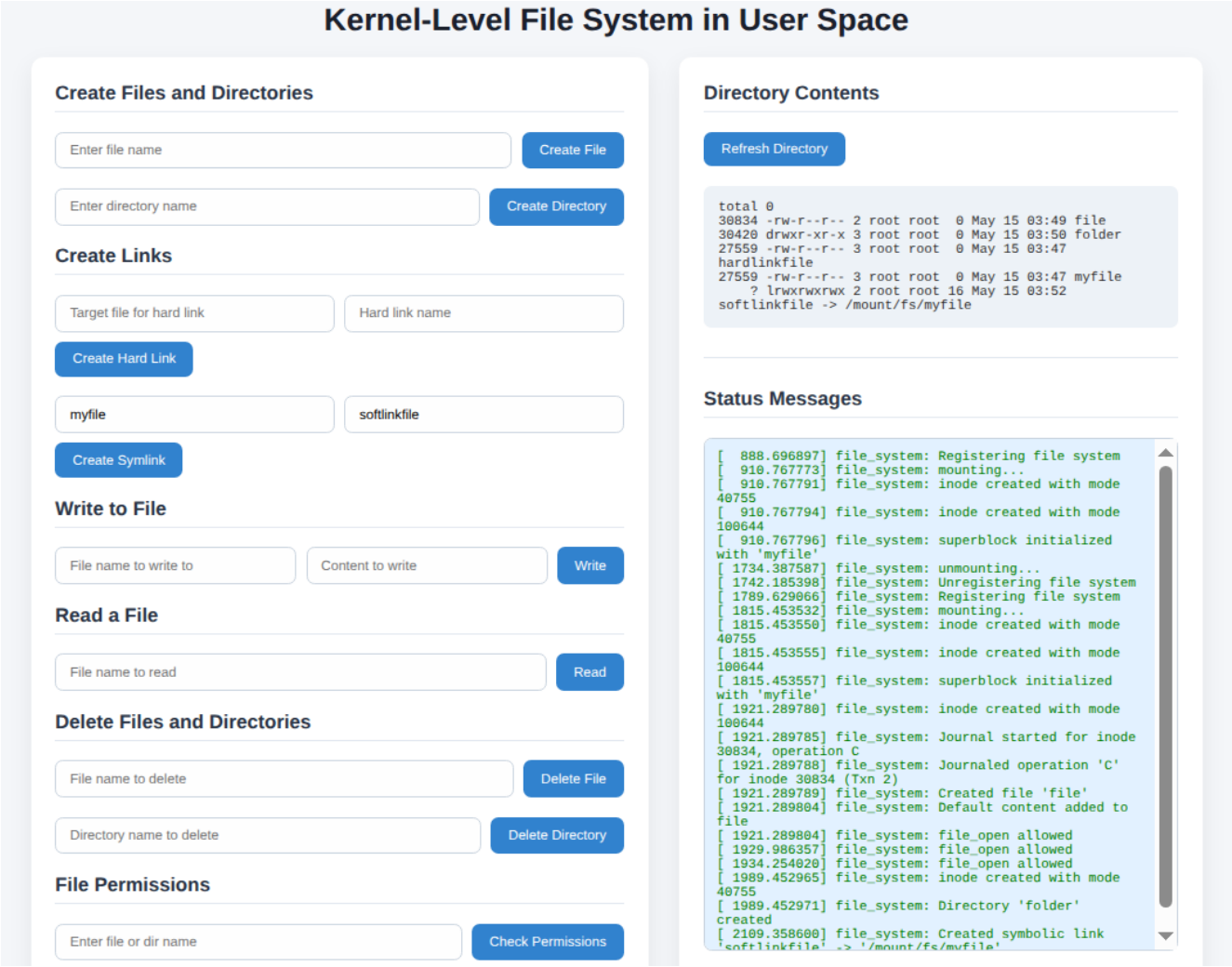

Figure 26: Directory content after soft link creation and refresh.

18.5.11 Read File Status Message and Content

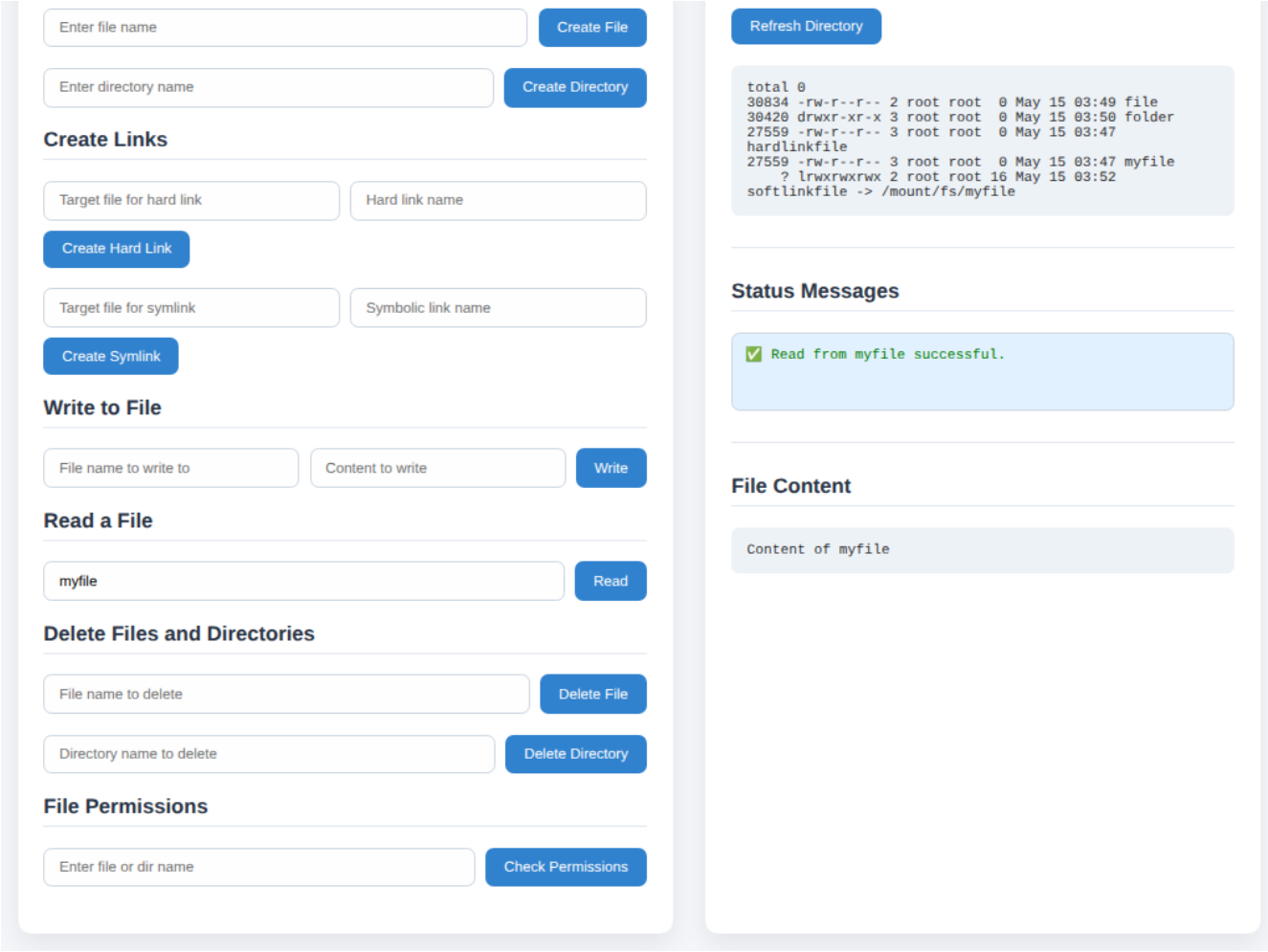

Figure 27: Status message after reading file and its content.

18.5.12 Write to File Status Message

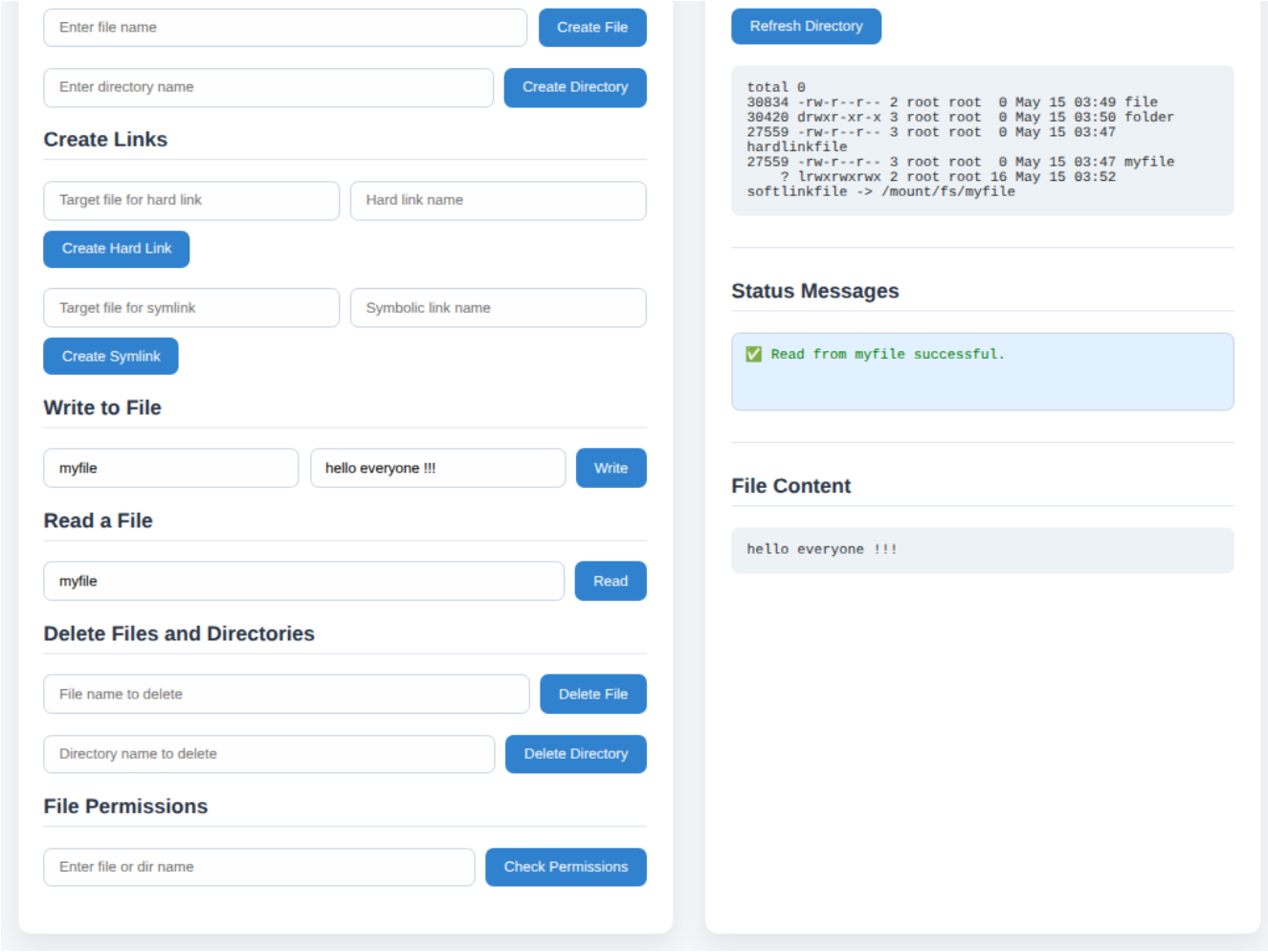

Figure 28: Status message after writing to file.

18.5.13 Directory Content Refresh After Deleting File

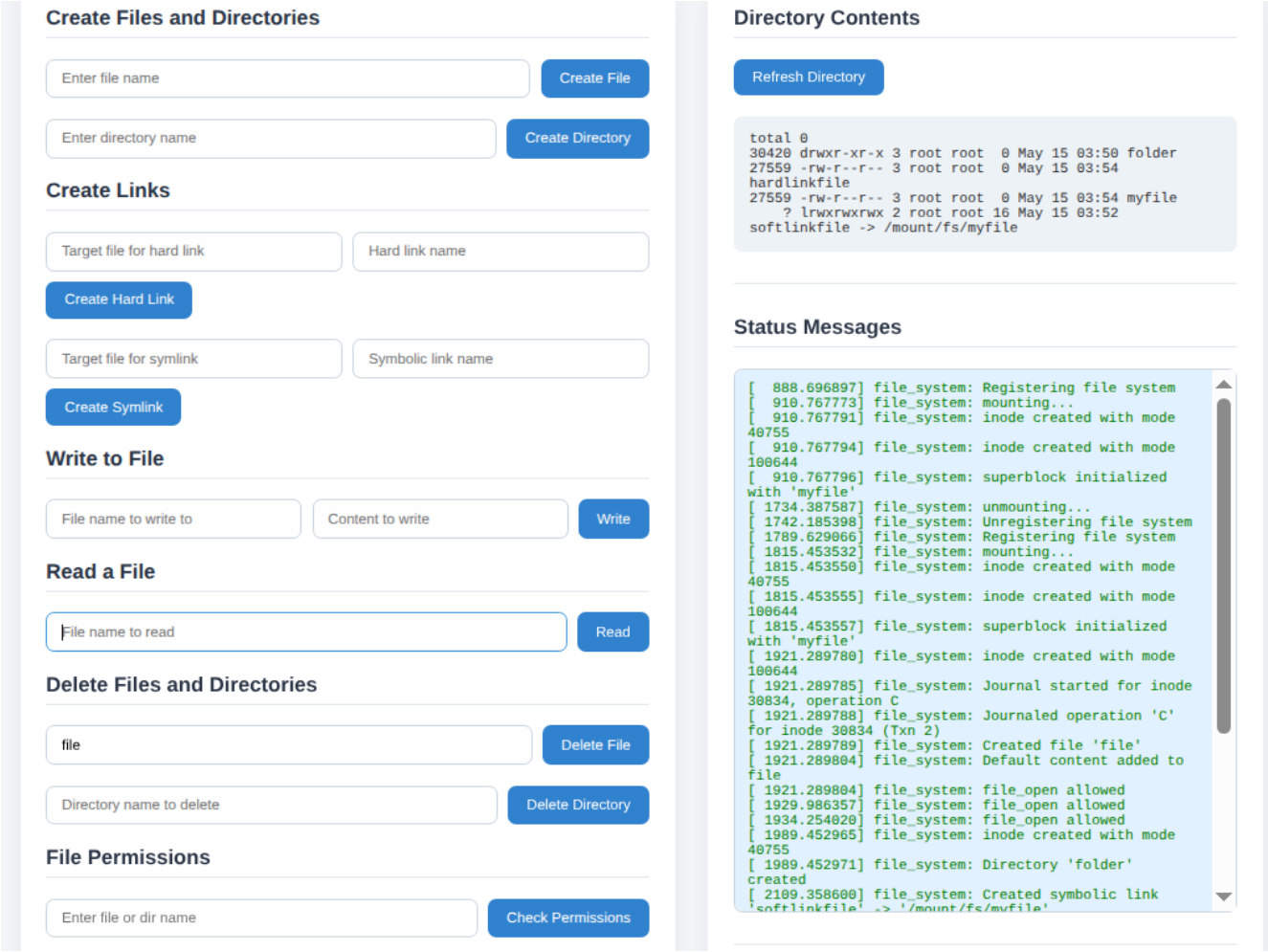

Figure 29: Directory content after file deletion.

18.5.14 Directory Content Refresh After Deleting Folder

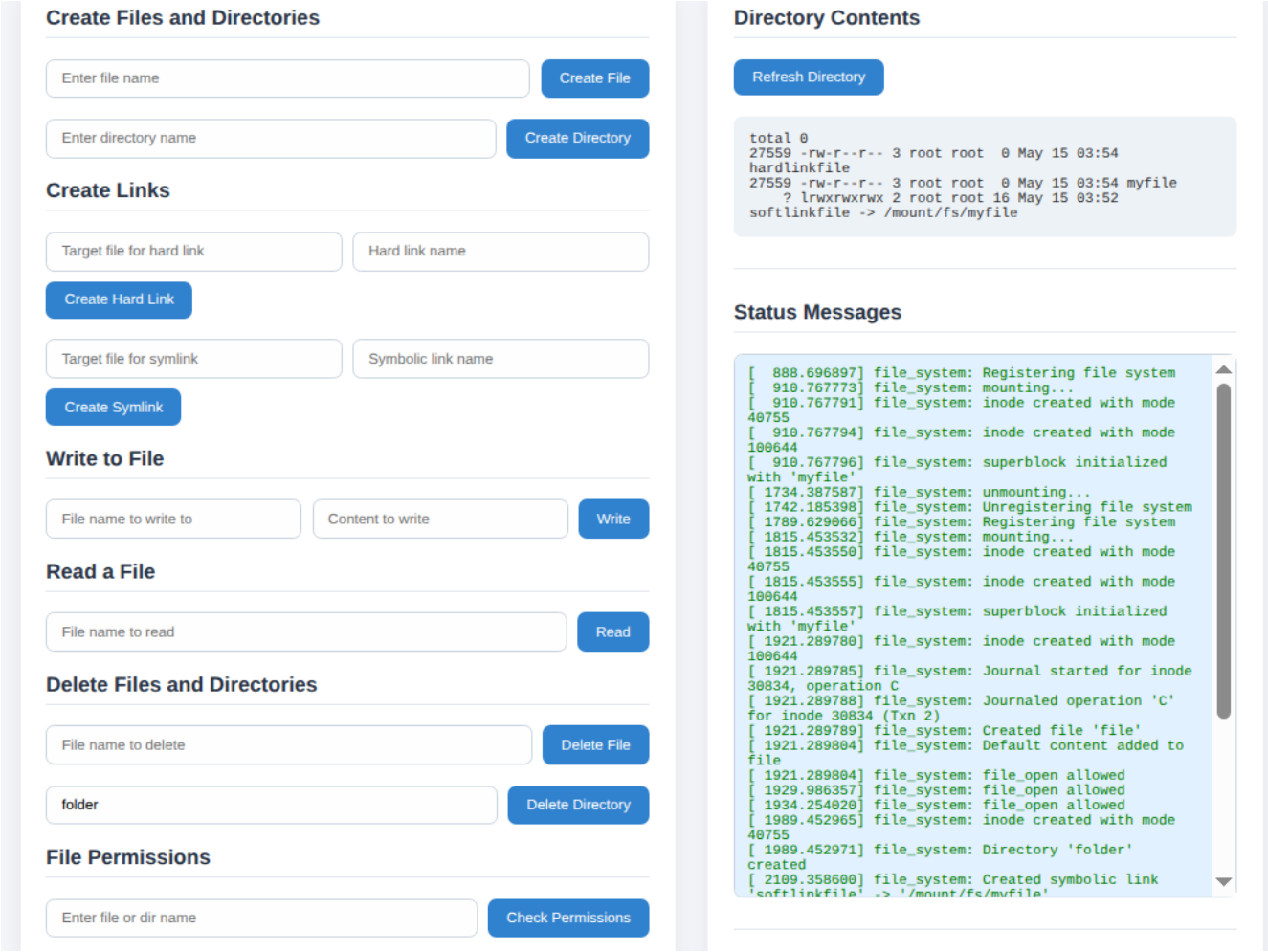

Figure 30: Directory content after folder deletion.